Improve Language Modelling for Code Completion by Tree Language Model with Tree Encoding of Context

Yixiao Yang
School of Software
Tsinghua University
Beijing, China
yangyixiaofirst@163.com

Xiang Chen
Beijing, China
kuailezhish@gmail.com

Jiaguang Sun
School of Software
Tsinghua University
Beijing, China

***Abstract*—In last few years, using a language model such as LSTM to train code token sequences is the state-of-art to get a code generation model. However, source code can be viewed not only as a token sequence but also as a syntax tree. Treating all source code tokens equally will lose valuable structural information. Recently, in code synthesis tasks, tree models such as Seq2Tree and Tree2Tree have been proposed to generate code and those models perform better than LSTM-based seq2seq methods. In those models, encoding model encodes user-provided information such as the description of the code, and decoding model decodes code based on the encoding results of user-provided information. When applying decoding model to decode code, current models pay little attention to the context of the already decoded code. According to experiments, using tree models to encode the already decoded code and predicting next code based on tree representations of the already decoded code can improve the decoding performance. Thus, in this paper, we propose a novel tree language model (TLM) which predicts code based on a novel tree encoding of the already decoded code (context). The experiments indicate that the proposed method outperforms state-of-arts in code completion.**

***Index Terms*—code completion, tree language model, tree encoding of context**

I. Introduction

Taking highly repetitive and predictable [1] source code as natural languages to build language models (n-gram, n-gram with Bayes or decision tree, RNN, LSTM) to suggest code fragments has made great progress [2]–[9]. However, sequential models does not explicitly model the tree structure. In the meanwhile, tree classification models such as TreeNN [10] and TBCNN [11] have more powerful ability in capturing the characteristics of trees. The works in [10] have confirmed that tree models perform better than sequential models in classification tasks. On the other hand, it is hard to adapt the code classification models as the code generation models. This paper addresses this challenging problem and demonstrates the special property of the proposed model.

In code synthesis models, encoding module is used for encoding the user-provided information, decoding module is used for decoding the desired code from the encoding of the provided information. In decoding module, we creatively use encoding model to encode the already decoded code to improve the performance of the decoding model. A novel tree decoding model (tree language model) which consists of Encoding model and Decoding model is proposed. In the rest of this paper, the term encoding model refers to the model which is used for encoding the already decoded code (context) in decoding procedure, not the one used for encoding the provided information in code synthesis tasks. Encoding model is responsible for generating the representation for a tree or a sub-tree. Decoding model is designed to traverse the syntax tree to accumulate encoding of encountered sub-trees to predict the next code. We propose a novel encoding model based on two-dimensional LSTM to generate better representations for trees. The experiments indicate that the proposed model outperforms state-of-arts. In summary, the contributions of this paper include: 1) A framework is proposed to predict next code based on the accumulated representations of sub-trees in an AST. 2) A new encoding model based on two-dimensional LSTM is designed to generate representations for trees.

Related Work: The statistical n-gram language model has been widely used in capturing patterns of source code. The n-gram model was applied to lexical code tokens in [1]. In SLAMC [2], they improved n-gram by associating code tokens with topics. In cacheca [4], they improved n-gram with caching recently appeared tokens in local files to improve prediction performance. In [12], source code was abstracted into DSL and had been sampled and validated until the good code suggestion was obtained. Deep learning techniques such as RNN, LSTM were applied to code generation model [6] [8] [9] to achieve a higher prediction accuracy. Decision tree was applied to code suggestion relying on a hybrid language model presented by [7]. In [5], code was trained on graphs. Naive-Bayes was integrated into n-gram model to suggest API usages. Seq2Seq [13], Seq2Tree [14] models were proposed to translate text description into the abstract syntax tree (AST) to handle the problem of code synthesis. The work [15] used basic Seq2Seq model to synthesize the API usage patterns based on natural languages. The aims of language model and the decoding

DOI reference number: 10.18293/SEKE2019-057

module in code synthesis models are similar: to generate the most likely next code. Thus, we compare these methods in the experiments. In the problem of program translation, Tree2Tree [16] model is proposed.

II. Tree Language Model

A. Preliminary

AST(Abstract Syntax Tree): Figure 1 shows an example of an expression and its corresponding AST. In this figure, the tree is placed horizontally: the root is on the left and the leaves are on the right. In rest of this paper, all trees are placed horizontally.

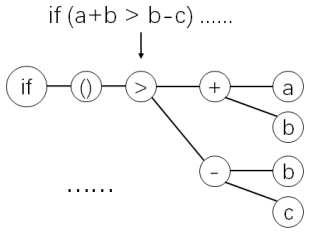


Fig. 1. an example of AST

Concepts on AST: For the abstract syntax tree of source code, some concepts need to be given.

- τ_n: the content (token) of node n is τ_n
- Sibling nodes: If two nodes have the same parent, they are siblings. For example, in Figure 1, node a and node b are sibling nodes, node $+$ and node $-$ are sibling nodes.
- Previous sibling node: If node m appears prior to node n and m and n are sibling nodes, m is called the previous sibling of n. For example, in Figure 1, node a is located above node b, so a is the previous sibling of b.
- Previous adjacent sibling node: If node m is the previous sibling of node n and m, n are adjacent, node m is called the previous adjacent sibling of node n. For example, in Figure 4, n is the previous adjacent sibling of o.
- Sibling trees: If roots of two sub-trees are siblings, these two sub-trees are called sibling trees. For example, in Figure 4, the sub-tree rooted at m and the sub-tree rooted at n are sibling trees, sub-tree rooted at node m and the sub-tree rooted at node o are sibling trees.
- Previous sibling tree: If the root of tree T_1 is the previous sibling of the root of tree T_2, tree T_1 is the previous sibling tree of tree T_2. For example, in Figure 4, the tree rooted at node m is the previous sibling tree of the tree rooted at node n.

LSTM: In this paper, the main logics of LSTM and two-dimensional LSTM (2DLSTM) will be expressed as two functions: LSTM and 2DLSTM. These two functions are shown in Figure 2. Function LSTM(x, $cell$, h) takes three inputs while the function 2DLSTM(x, $cell$, h, $cell2$, $h2$) takes five inputs. The outputs for this two functions are same: $cell_{new}$, h_{new}. In this paper, parameter x is the embedding of node token on abstract syntax tree of source code. In LSTM, $cell$, h pair is taken as accumulated information from one direction, and 2DLSTM receives accumulated information: $cell$, h; $cell2$, $h2$ from two directions. Please consult [17] for more information about LSTM and two-dimensional LSTM.

Algorithm LSTM(x, $cell$, h)

Require: x, $cell$, h
Ensure: $cell_{new}$, h_{new}

$$i = \sigma(W_i \cdot [x, h] + b_i)$$
$$j = tanh(W_j \cdot [x, h] + b_j)$$
$$f = \sigma(W_f \cdot [x, h] + b_f)$$
$$o = \sigma(W_o \cdot [x, h] + b_o)$$
$$cell_{new} = f * cell + i * j$$
$$h_{new} = o * tanh(cell_{new})$$

return $cell_{new}$, h_{new}

Algorithm 2DLSTM(x, $cell$, h, $cell2$, $h2$)

Require: x, $cell$, h, $cell2$, $h2$
Ensure: $cell_{new}$, h_{new}

$$i = \sigma(W_i \cdot [x, h, h2] + b_i)$$
$$j = tanh(W_j \cdot [x, h, h2] + b_j)$$
$$f1 = \sigma(W_{f1} \cdot [x, h, h2] + b_{f1})$$
$$f2 = \sigma(W_{f2} \cdot [x, h, h2] + b_{f2})$$
$$o = \sigma(W_o \cdot [x, h, h2] + b_o)$$
$$cell_{new} = f1 * cell + f2 * cell2 + i * j$$
$$h_{new} = o * tanh(cell_{new})$$

return $cell_{new}$, h_{new}

Fig. 2. LSTM and two-dimensional LSTM

TreeNN: Given a tree T, TreeNN [10] generates vector representation of a tree. TreeNN is defined in Algorithm 1. In Algorithm 1, the symbols: c_1, c_2, ..., c_k represent all k children of node n, $[\cdot, \cdot]$ is the vector concatenation operator, σ is the activation function such as tanh.

Algorithm 1 TreeNN(node n)

Require: node n
Ensure: representation of tree rooted at node n

if n **is** a leaf **then**
  h = embedding of τ_n
else
  $h = \sigma(W_{\tau_n} \cdot [\text{TreeNN}(c_1), ..., \text{TreeNN}(c_k)])$
end if
return h

Additional Functions Other functions used are defined here.

- GetLastChildOfNode(n): returns the last child of tree node n. If n does not have any child, function returns *null*. In Figure 1, node c is the last child of node $-$, GetLastChildOfNode($-$) returns c.
- GetPrevAdjacentSiblingOfNode(n): returns previous adjacent sibling of node n, if n does not have a previous sibling (n is the first child of its parent), returns *null*. In Figure 1, node $+$ is the previous adjacent sibling of node $-$, then GetPrevAdjacentSiblingOfNode($-$) returns $+$.
- Parent(n): returns the parent node of node n in AST.

B. Encoding model

Encoding model such as TreeNN generates the representation for a tree through visiting the tree in post-order traversal. Figure 3 gives an illustration of the execution flow of encoding model. We propose the novel encoding model based on two-dimensional LSTM: AccuredTreeGen model. AccuredTreeGen model is the one used in Decoding model of Tree Language Model. The aim of AccuredTreeGen(n) is to generate the representation for a set of trees: the tree T rooted at node n and all previous sibling trees of tree T. Unlike

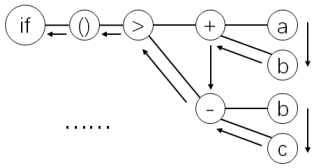
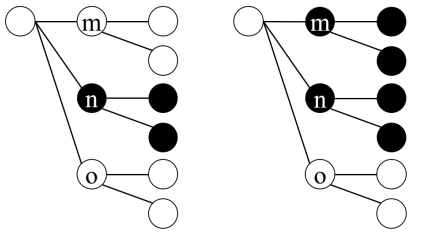


Fig. 3. data flow of encoding

traditional tree encoding model which generates encoding for a tree, The AccuredTreeGen model generates encoding for a set of trees. Figure 4 illustrates the difference of trees



(a) nodes handled by **TreeNN**(n) (b) nodes handled by **AccuredTreeGen**(n)

Fig. 4. An illustration of TreeNN and AccuredTreeGen

handled by TreeNN(n) and AccuredTreeGen(n). The black circles are nodes in trees to be processed. TreeNN(n) only handles the tree rooted at node n while AccuredTreeGen(n) extraly handles all previous sibling trees of the tree handled by TreeNN(n).

Algorithm 2 AccuredTreeGen(node n)

Require: node n
Ensure: representation of tree with previous sibling trees
  if n == *null* **then**
    return $cell_{zero}$, h_{zero}
  end if
  $embed_{\tau_n}$ = embedding of τ_n
  $sibling_{prev}$ = GetPrevAdjacentSiblingOfNode(n)
  $child_{last}$ = GetLastChildOfNode(n)
  $cell$, h = AccuredTreeGen($sibling_{prev}$)
  $cell2$, $h2$ = AccuredTreeGen($child_{last}$)
  $cell_{new}$, h_{new} = 2DLSTM($embed_{\tau_n}$, $cell$, h, $cell2$, $h2$)
  return $cell_{new}$, h_{new}

The algorithm of AccuredTreeGen is defined in Algorithm 2. AccuredTreeGen is a recursive model. For a node n, apart from the embedding of node n, the result of AccuredTreeGen(n) depends on the result of AccuredTreeGen($previous\ adjacent\ sibling\ of\ n$) and the result of AccuredTreeGen($last\ child\ of\ n$). Recursively, For the last child: $child_{last}$ of node n, the result of AccuredTreeGen($child_{last}$) depends on the result of AccuredTreeGen($previous\ adjacent\ sibling\ of\ child_{last}$) and the result of AccuredTreeGen($last\ child\ of\ child_{last}$). Keep computing the dependency recursively, we will find that the result of AccuredTreeGen(n) is the accumulated information of all nodes on a set of trees: tree T rooted at node n and all previous sibling trees of tree T. If node n is *null*, AccuredTreeGen(n) will return $cell_{zero}$ and h_{zero} which are fixed default zero values. This is also the termination for the recursive AccuredTreeGen model. The gates used in two-dimensional LSTM make the model less troubled by vanishing gradient problem than TreeNN.

C. Decoding model

Decoding model traverses from the root to the leaves on a tree in pre-order to predict the token of each node. Figure

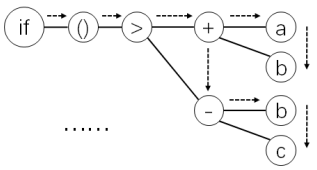


Fig. 5. data flow of decoding

5 illustrates the execution flow of the Decoding model. The Decoding model contains two sub-models: DecodeFirstChild model and DecodeNextSibling model. When we are visiting node n in pre-order traversal, DecodeFirstChild(n) generates prediction information for predicting the first child of node n. DecodeNextSibling(n) generates prediction information for predicting the next sibling of node n. The information generated by DecodeFirstChild or DecodeNextSibling consists of two vectors ($cell$, h). Every node except the root is either the first child or the next sibling of some node. So each node except the root can receive the prediction information from its parent or its previous adjacent sibling. The root node receives the fixed default values. The function FetchPrediction(node n) is defined to get the prediction information generated for predicting node n. The definition is in Algorithm 3.

Algorithm 3 FetchPrediction(node n)

Require: node n
Ensure: ($cell$, h) for predicting node n
  if n **is** root of AST **then**
    return $cell_{zero}$, h_{zero}
  end if
  $parent_n$ = Parent(n)
  if n **is** the first child of $parent_n$ **then**
    $cell$, h = DecodeFirstChild($parent_n$)
  else
    $sibling_{prev}$ = GetPrevAdjacentSiblingOfNode(n)
    $cell$, h = DecodeNextSibling($sibling_{prev}$)
  end if
  return $cell$, h

In Algorithm 3, if node n is the root of AST which means node n does not have parent or previous siblings, the default zero values: $cell_{zero}$ and h_{zero} are returned. If node n is the first child of node $parent_n$, DecodeFirstChild($parent_n$)

generates prediction information ($cell$, h) for predicting the content of node n, as shown in the *then* branch of the if-statement in Algorithm 3. If node n is not the first child of parent $parent_n$, in this case, node n must have previous adjacent sibling: $sibling_{prev}$, DecodeNextSibling($sibling_{prev}$) generates prediction information ($cell$, h) for n, as shown in the *else* branch of the if-statement in Algorithm 3. In summary, FetchPrediction(n) just fetches prediction information from the parent of n or the previous sibling of n according to the position of n in AST. Assume that n is the first child of $parent_n$, then, the information ($cell$, h) returned by FetchPrediction(n) is just the information ($cell$, h) returned by DecodeFirstChild($parent_n$). Note that the prediction ($cell$, h) for node n returned by FetchPrediction(n) can be taken as the accumulated information of nodes visited before n.

The algorithm of DecodeFirstChild model is defined in algorithm 4. The embedding of node n and the accumulated

Algorithm 4 DecodeFirstChild(node n)

Require: node n
Ensure: ($cell_{out}$, h_{out}) for first child of node n
  $cell$, h = FetchPrediction(n)
  $embed_{\tau_n}$ = embedding of τ_n
  $cell_{out}$, h_{out} = LSTM($embed_{\tau_n}$, $cell$, h)
  return $cell_{out}$, h_{out}

information of nodes visited before n ($cell$, h returned by FetchPrediction(n)) are fed into LSTM to predict the first child of node n. The FetchPrediction, DecodeFirstChild and DecodeNextSibling (described in the following) functions call each other and form a recursive neural model. To predict the sibling of a node n, the algorithm of DecodeNextSibling model is in Algorithm 5. If $child_{last}$ is the last child of node n, AccuredTreeGen($child_{last}$) generates the representation ($cell2$, $h2$) for a set of trees: tree $T_{lastchild}$ rooted at node $child_{last}$ and all previous sibling trees of tree $T_{lastchild}$. The nodes in tree $T_{lastchild}$ and all previous sibling trees of $T_{lastchild}$ constitute all the descendants of node n. The embedding of node n, the accumulated information of nodes visited before n ($cell$, h returned by FetchPrediction(n)) and the representation ($cell2$, $h2$) for all descendants of node n are fed into two-dimensional LSTM to predict next sibling of node n. For a node n, DecodeForFirstChild(n)

Algorithm 5 DecodeNextSibling(node n)

Require: node n
Ensure: ($cell_{out}$, h_{out}) for next sibling of node n
  $cell$, h = FetchPrediction(n)
  $child_{last}$ = GetLastChildOfNode(n)
  $cell2$, $h2$ = AccuredTreeGen($child_{last}$)
  $embed_{\tau_n}$ = embedding of τ_n
  $cell_{out}$, h_{out} = 2DLSTM($embed_{\tau_n}$, $cell$, h, $cell2$, $h2$)
  return $cell_{out}$, h_{out}

and DecodeForNextSibling(n) both use the embedding of node n. The difference between DecodeForFirstChild(n) and DecodeForNextSibling(n) is whether or not to take all descendants of node n into consideration. When predicting the first child of node n, we do not need to take descendants of node n into consideration because no descendant of node n has been visited in pre-order traversal of AST. When predicting the sibling of node n, all descendants of this node have been visited (predicted) and we use AccuredTreeGen to explicitly encode the already visited (predicted) sub-trees. Given a tree, starting with the root of that tree, all nodes can be predicted by keeping inferring the first child of a node and the next sibling of a node. Figure 6 gives an illustration about the data flow of encoding model and decoding model to show how decoding model interacts with encoding model. In Figure 6, solid arrow means the data flow of encoding while dotted arrow means the data flow of decoding. In Figure 6, we use the content of a node to refer to that node. As shown in the figure, the nodes (), $>$ and $+$ are the first child of their parents. DecodeFirstChild model is used to generate prediction information for those nodes. To predict node $-$ which is the next sibling of node $+$, the embedding of token $+$, the prediction for node $+$ (the data flow is marked with dotted arrow), the representation for all descendants of node $+$ (the data flow is marked with solid arrow) will be fed into two-dimensional LSTM to generate a new $cell$, h.

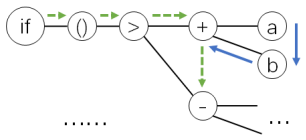


Fig. 6. an example of decoding combined with encoding

Tree Language Model is also a generalized framework in which the encoding model can be replaced with existing tree classification models. Take TreeNN as an example. If we want to use TreeNN in Tree Language Model, DecodeNextSibling model should be replaced with DecodeNextSiblingUsingTreeNN model. The definition of DecodeNextSiblingUsingTreeNN is in Algorithm 6. The difference between DecodeNextSibling and DecodeNextSiblingUsingTreeNN is that DecodeNextSiblingUsingTreeNN uses TreeNN to encode the root node n and all its descendants into one vector instead of taking apart them. LSTM instead of two-dimensional LSTM is applied. It is a pioneering work to adapt the tree classification model as a language model. Other tree models such as TBCNN and EQNET can be adapted in a similar way.

Algorithm 6 DecodeNextSiblingUsingTreeNN(node n)

Require: node n
Ensure: ($cell_{out}$, h_{out}) for next sibling of node n
  $cell$, h = FetchPrediction(n)
  $encoding_{tree}$ = TreeNN(n)
  $cell_{out}$, h_{out} = LSTM($encoding_{tree}$, $cell$, h)
  return $cell_{out}$, h_{out}

D. Predicting and Training

For every node n, the prediction result ($cell \in \mathbf{R}^d$, $h \in \mathbf{R}^d$) for node n returned by FetchPrediction(n) is used to compute the probability distribution of all tokens. The Algorithm 7 is the definition of function Predict. In Algorithm 7, $W_1 \in \mathbf{R}^{t \times d}$ and $bias \in R^t$ are model parameters, t represents the total number of unique tokens in data set, d is the length of the embedding vector for one token. The $probs$ is the probability distribution for all tokens. The top k elements with the highest probabilities will be the final complement result. Top-k accuracy is computed in the way that if the desired token appears in the top k recommended tokens, the prediction is right, otherwise, the prediction is wrong.

Algorithm 7 Predict(node n)

$cell$, h = FetchPrediction(n)
$logits = tanh(W_1 \cdot h + bias)$
$probs = softmax(logits)$
return $probs$

For every node n, training is to maximize the probability of n. This is achieved by minimize the loss of node n which is computed by function ComputeLoss. The Algorithm 8 is the definition of function ComputeLoss. In Algorithm 8, the $probs$ is the probability distribution for all tokens returned by Predict(n). The $probs[\tau_n]$ means choosing the probability of τ_n (the actual content of node n) from $probs$. The final loss is the summation of loss computed by ComputeLoss for each node in each AST. The training of the whole model is to minimize the final loss.

Algorithm 8 ComputeLoss(node n)

$probs$ = Predict(n)
$loss = -log(probs[\tau_n])$
return $loss$

III. Implementation

Source code [18] of all models along with all data sets has been public. The source code is parsed into AST through Eclipse JDT. The implementation is based on Deep learning platform: TensorFlow. The model parameters consists of parameters in LSTM, parameters in 2DLSTM and W_1, W_2 in Algorithm 7. Because TensorFlow does not offer the implementation of two-dimensional LSTM, the logic of two-dimensional LSTM is implemented by ourselves. The physical environment is the computer with Windows 10 64 bit OS, Intel i7-6850k CPU, 32G memory and one Geforce GTX 1080 Ti GPU. The learning rate and the momentum are automatically decided by Adam optimizer in TensorFlow. Global norm is used to clip the gradient. Examples are trained or tested one by one. The representation size (alias as embedding size or feature size) for one token is 128. We will keep training the model until the prediction accuracy on the validation set does not exceed the optimal value for 50 epochs.

IV. Experiment

Without loss of generality, one of the most widely used programming language Java is chosen to conduct experiments. Java projects with high number of stars on GitHub are extracted and filtered into different data sets. Source code in each data set is divided into training set, validation set and test set in the proportions 60%, 15%, 25%. Every function declared in Java files will be parsed into an abstract syntax tree and every tree node in AST will be predicted to compute the prediction accuracy. The tree will be flattened into a sequence to apply sequential models such as LSTM. The sequence is generated by traversing the tree in pre-order and appending the encountered node back to the sequence. We mark the 0.15% least frequently occurred code tokens in training set and all unseen tokens in validation set or test set as UNK.

Date Sets: Three data sets: Dataset A, Dataset B, Dataset C are collected to conduct experiments to examine the performance of models. Details are shown in the following Table. Dataset A consists of all java files in project *apache commons io* on Github. The project *google guava* is a Java project marked with 26554 stars on Github. Dataset B consists of Java files whose size is larger than 8K Bytes in the main module of *google guava*. Dataset C are Java files whose size is larger than 40K bytes in projects *Activiti* (3909 stars), *JComicDownloader* (188 stars), *ESPlorer* (733 stars). and *AbFab3D* (59 stars). The evaluation results on Dataset C may truly reflect the ability of each model because abstract syntax trees from Dataset C are huge. The fourth column headed by *Vocabulary* in Table IV means the quantity of unique tokens (the content of node on AST) on the data set.

	From Projects	Size	Vocabulary
Dataset A	apache commons io	2.0MB	5807
Dataset B	google guava	7.7MB	7538
Dataset C	Activiti & ESPlorer & AbFab3D & JComicDownloader	8.2MB	39886

Evaluation: In this section, Tree Language Model using the newly proposed encoding model: AccuredTreeGen model (based on two-dimensional LSTM) is abbreviated into TLM. Tree language model using TreeNN as the encoding model is abbreviated into TLM-TNN. LSTM and the decoding module in Tree2Tree [16] are also included in the baselines. The top-k accuracy (value is in percentage, % is omitted to save the space) is computed on every node in tree and the final top-k accuracy is the average of top-k accuracy of all nodes in all trees. When predicting next node, the model computes the probabilities for all candidate tokens. If we rank all tokens according to probabilities from large to small. For example, the token with the highest probability ranks 1, the token with the second highest probability ranks 2. The value in column headed with *mrr* means the average of the reciprocal of the rank for each token in a data set. This metric indicts the overall prediction performance of the model. The larger the mrr, the better the model. The column headed with *enpy* shows the entropy (log_2 value of the perplexity). The smaller the entropy, the better the model.

Table I is the evaluation result of different models on test set. On small data sets: DS1, TLM achieves 22.2%, 9.3% higher top-1 prediction accuracy than tree decoding module in Tree2Tree and LSTM. The performance of TLM and TLM-TNN is similar, for top-1 accuracy and mrr, TLM performs better. On large data set: DS2, Tree2Tree performs much worse than other three models. TLM performs the best in all measurements. As can be shown, if the models used for code synthesis are directly applied to code completion tasks, the performance is bad. The reason will be described later. On large data set DS3, TLM achieves 40.3%, 13.8% higher top-1 accuracy than tree decoding module in Tree2Tree and LSTM. The performance of TLM and TLM-TNN is similar, for top-1 accuracy, TLM performs better. For top-k accuracy, as k becomes larger, performances of all models tend to be closer to each other. For top-10 accuracy, performances of all models are close. This indicates that for the top-k accuracy, the smaller the k, the more effective is the top-k accuracy to illustrate performances of different models. As can be seen, both TLM and TLM-TNN performs better than the rest non-TLM models. This demonstrates the advantages of the newly proposed Tree Language Model framework which uses tree encodings to encode the context to help predict code. In future work, we may adopt other measurements to investigate different models. By considering all conditions, on all 3 data sets, Tree Language Model framework performs better than other models. TLM performs the best on top-1 accuracy.

TABLE I
PREDICTION ACCURACY ON TEST SET

		top-1	top-3	top-6	top-10	mrr	enpy
A	Tree2Tree	38.6	57.4	63.7	67.6	0.49	4.3
	TLM-TNN	46.1	61.4	66.5	69.3	0.54	4.0
	LSTM	43.2	57.5	63.7	66.7	0.51	4.2
	TLM	**47.2**	61.4	66.3	68.9	0.55	4.1
		top-1	top-3	top-6	top-10	mrr	enpy
B	Tree2Tree	39.3	60.8	70.3	75.6	0.52	3.5
	TLM-TNN	68.6	81.3	85.8	87.9	0.76	2.6
	LSTM	70.1	80.7	84.6.8	86.6	0.76	3.0
	TLM	**72.1**	82.1	86.0	87.8	0.78	2.4
		top-1	top-3	top-6	top-10	mrr	enpy
C	Tree2Tree	34.7	54.0	61.7	66.1	0.46	6.2
	TLM-TNN	46.2	60.5	65.9	68.0	0.53	5.8
	LSTM	42.9	57.9	63.8	67.1	0.51	5.9
	TLM	**48.8**	60.8	65.2	67.5	0.53	5.8

The reason for better performance of Tree Language Model framework is adopting tree encoding model to capture the characteristics of encountered trees or sub-trees. In Tree2Tree, a tree must be converted to a binary tree. This step introduces extra nodes. The extra introduced nodes make predicting more difficult. When standing at a node, two LSTM models are used for predicting the left child node and right child node of that node separately at the same time. Predicting left without using information of right (vice versa) causes the performance declining. Current synthesis models pay much attention to predicting code based on user-provided information but pay little attention to predicting next code based on the already decoded code (context). This paper investigates this problem.

V. CONCLUSION

This paper proposed a novel tree language model consisting of decoding model and encoding model. Two-dimensional LSTM is adopted to deal with the structural characteristics of trees. The experiments demonstrate that tree language model (TLM) achieves better top-1 prediction accuracy on large data set compared to state-of-art models.

REFERENCES

[1] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 837–847. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2012.6227135

[2] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 532–542. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458

[3] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, p. 44. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[4] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *The ACM Sigsoft International Symposium*, 2014, pp. 269–280.

[5] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 858–868. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.336

[6] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Ieee/acm Working Conference on Mining Software Repositories*, 2015, pp. 334–345.

[7] V. Raychev, P. Bielik, and M. T. Vechev, "Probabilistic model for code with decision trees," in *OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016, pp. 731–747. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984041

[8] H. K. Dam, T. Tran, and T. T. M. Pham, "A deep language model for software code," in *FSE 2016: Proceedings of the Foundations Software Engineering International Symposium*. [The Conference], 2016, pp. 1–4.

[9] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[10] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1631–1642.

[11] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 1287–1293.

[12] V. Raychev, P. Bielik, M. T. Vechev, and A. Krause, "Learning programs from noisy data," in *POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 761–774. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837671

[13] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[14] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[15] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 631–642. [Online]. Available: https://doi.org/10.1145/2950290.2950334

[16] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *arXiv preprint arXiv:1802.03691*, 2018.

[17] A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer Berlin Heidelberg, 2012.

[18] "The source code of models in the experiments and all data sets," https://www.dropbox.com/s/8typta3a81htclr/TLM.zip.